MASSACHUSETTS INTITUTE OF TECHNOLOGY VLSI PUBLICATIONS

VLSI Memo No. 89-570
October 1989

A Unified Approach to the Synthesis of Fully Testable Sequential Machines

Srinivas Devadas and Kurt Keutzer

Abstract

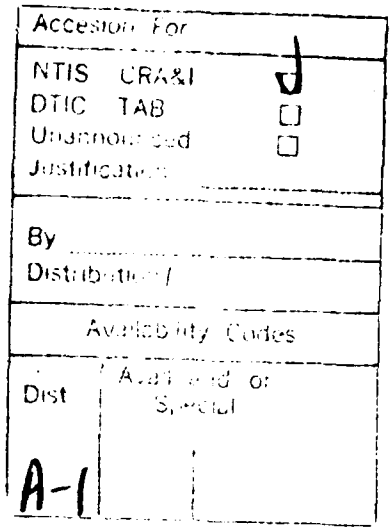
In this paper we attempt to unify and extend the various approaches to synthesizing fully testable sequential circuits that can be modeled as finite state machines (FSMs). We first identify classes of redundancies and isolate *equivalent-state redundancies* as those most difficult to eliminate. We then show that the essential problem behind equivalent-state redundancies is the creation of valid/invalid state pairs. We devote the remainder of the paper to techniques for developing *differentiating sequences* for valid/invalid state pairs created by a fault, as well as to techniques for retaining these sequences in the presence of that fault.

A variety of techniques have been proposed to address this problem. At one end of the spectrum there are optimal synthesis procedures that ensure full testability by eliminating redundancies via the use of appropriate don't care sets. At the other end of the spectrum there are constrained synthesis procedures that produce fully and easily testable sequential circuits by restricting the implementation of the logic. The optimal synthesis procedures require fewer constraints on the logic but increase the expense of logic optimization to the point that CPU time requirements may be unacceptable. The constrained synthesis procedures require relatively simple logic optimization procedures but constrain the logic to the point that the area penalty may be unacceptable.

In this paper we use the notion of *fault-effect disjointness* to explore the landscape between these two boundaries and demonstrate a spectrum of methods that place relatively more-or-less emphasis on either logic optimization or constrained synthesis. Techniques used in this exploration include fault simulation, Boolean covering, algebraic factorization and state assignment.

We present experimental results using the new synthesis procedures as well as comparisons to previous approaches.

Microsystems Technology Laboratories | Massachusetts Institute of Technology | Cambridge Massachusetts 02139 | Room 39-321 Telephone (617) 253-0292



Acknowledgements

This research was supported in part by the Defense Advanced Research Projects Agency under contract N00014-87-K-0825.

Author Information

Devadas: Department of Electrical Engineering and Computer Science, Room 36-848, MIT, Cambridge, MA 02139. (617) 253-0454.

Keutzer: AT & T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974. (201) 522-6332.

A Unified Approach to the Synthesis of Fully Testable Sequential Machines

Srinivas Devadas
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology, Cambridge

Kurt Keutzer
AT&T Bell Laboratories, Murray Hill, New Jersey

Abstract

In this paper, we attempt to unify and extend the various approaches to synthesizing fully testable sequential circuits that can be modeled as finite state machines (FSMs). We first identify classes of redundancies and isolate *equivalent-state redundancies* as those most difficult to eliminate. We then show that the essential problem behind equivalent-state redundancies is the creation of valid/invalid state pairs. We devote the remainder of the paper to techniques for developing *differentiating sequences* for valid/invalid state pairs created by a fault, as well as to techniques for retaining these sequences in the presence that fault.

A variety of techniques have been proposed to address this problem. At one end of the spectrum there are optimal synthesis procedures that ensure full testability by eliminating redundancies via the use of appropriate don't care sets. At the other end of the spectrum there are constrained synthesis procedures that produce fully and easily testable sequential circuits by restricting the implementation of the logic. The optimal synthesis procedures require fewer constraints on the logic but increase the expense of logic optimization to the point that CPU time requirements may be unacceptable. The constrained synthesis procedures require relatively simple logic optimization procedures but constrain the logic to the point that the area penalty may be unacceptable.

In this paper we use the notion of *fault-effect disjointness* to explore the landscape between these two boundaries and demonstrate a spectrum of methods that place relatively more-or-less emphasis on either logic optimization or constrained synthesis. Techniques used in this exploration include include fault simulation, Boolean covering, algebraic factorization and state assignment.

We present experimental results using the new synthesis procedures as well as comparisons to previous approaches.

1 Introduction

Can a sequential circuit be completely tested without adding scan logic? This is perhaps the most open problem in the area of testing. One natural approach to solving this problem is to improve current sequential test generation algorithms. The primary drawback to this approach is that circuit sizes are increasing so quickly that even significant improvements in sequential test generation algorithms cannot keep up. A radically different approach is synthesis for sequential testability. In this approach it is the structure of the circuit itself that is modified to produce fully testable designs.

The idea that logic synthesis and optimization can have a very profound effect on the testability of a synthesized combinational or sequential circuit has been recognized [6]. The relationship between testability and Boolean minimization for two-level combinational circuits dates back to the Quine-McCluskey algorithm [10]. Notions of prime implicants and irredundant covers are basic to all two-level Boolean minimization procedures and these imply immunity to stuck-at fault redundancies in two-level combinational circuits. Initial work in the area of multi-level logic synthesis and testability involved the use of implication procedures to eliminate redundancies in combinational logic circuits [2]. The relationships between redundancies and don't cares in combinational circuits was more thoroughly investigated in [1], where the notions of primality and irredundancy were generalized for multi-level circuits. Recent work in synthesis for testability has been able to ensure complete multiple fault testability for multi-level combinational logic circuits [9].

Relationships between sequential logic synthesis and non-scan sequential circuit testability are equally intimate. Scan logic appears to be less necessary for ensuring the testability of datapath portions of circuits because datapath portions have less feedback [11] [5]. As a result, the remaining challenges in synthesizing sequentially testable circuits are to synthesize fully/easily testable control portions and to combine these with datapath portions. Control portions are most commonly implemented as finite state machines (FSMs).

In this paper, we attempt to unify and extend the various approaches to synthesizing fully testable sequential circuits that can be modeled as finite state machines (FSMs). We first identify classes of redundancies and isolate *equivalent-state redundancies* as those most difficult to eliminate. We then show that the essential problem behind equivalent-state redundancies is the creation of valid/invalid state pairs. We devote the remainder of the paper to techniques for developing *differentiating sequences* for valid/invalid state pairs created by a fault, as well as to techniques for retaining these sequences in the presence that fault.

A variety of techniques have been proposed to address this problem. At one end of the spectrum there are optimal synthesis procedures that ensure full testability by eliminating redundancies via the use of appropriate don't care sets. At the other end of the spectrum there are constrained synthesis procedures that produce fully and easily testable sequential circuits by restricting the implementation of the logic. The optimal synthesis procedures require fewer constraints on the logic but increase the expense of logic optimization to the point that CPU time requirements may be unacceptable. The constrained synthesis procedures require relatively simple logic optimization procedures but constrain the logic to the point that the area penalty may be unacceptable.

In this paper we use the notion of *fault-effect disjointness* to explore the landscape between these two boundaries and demonstrate a spectrum of methods that place relatively more-or-less emphasis on either logic optimization or constrained synthesis. Techniques used in this exploration include include fault simulation, Boolean covering, algebraic factorization and state assignment.

Finally, we present experimental and analytical comparisons between various testability-driven synthesis procedures that provide insights as to the relative merits of the different procedures.

Basic definitions and terminology are given in Section 2. In Section 3, we review the types of sequential redundancies in FSMS. In Section 4 we describe general methods for removing some classes of redundancies and review theorems regarding unconditional testability of faults in sequential circuits. In Section 5, we present the notion of differentiating sequences and describe a generic synthesis procedure that results in fully testable sequential machines. We then present a unification of synthesis-for-testability approaches under the umbrella of a concept strongly related to differentiating sequences, fault-effect disjointness, and show that previous synthesis approaches can be viewed as special cases of the generic synthesis procedure. In addition, we describe new Boolean covering and algebraic factorization techniques that represent intermediate solutions to the problem of synthesizing fully testable sequential machines. Preliminary experimental results using the new synthesis procedure proposed here as well as comparisons to previous techniques are given in Section 6.

2 Preliminaries

A **variable** is a symbol representing a single coordinate of the Boolean space (*e.g.* a). A **literal** is a variable or its negation (*e.g.* a or $\overline{a}$). A **cube** is a set C of literals such that $x \in C$ implies $\overline{x} \notin C$ (*e.g.*, $\{a, b, \overline{c}\}$ is a cube, and $\{a, \overline{a}\}$ is not a cube). A cube represents the conjunction of its literals. The trivial cubes, written **0** and **1**, represent the Boolean functions 0 and 1 respectively. An **expression** is a set f of cubes. For example, $\{\{a\}, \{b, \overline{c}\}\}$ is an expression consisting of the two cubes $\{a\}$ and $\{b, \overline{c}\}$. An expression represents the disjunction of its cubes.

A cube may also be written as a bit vector or a set of variables with each bit position representing a distinct variable. The values taken by each bit can be 1, 0 or 2 (or – or don't care), signifying the true form,

negated form and non-existence respectively of the variable corresponding to that position. A **minterm** is a cube with only 0 and 1 entries.

A finite state machine (FSM) is represented by its **State Transition Graph** $G(V, E, W(E))$ where V is the set of vertices corresponding to the set of states S. An edge joins v_i to v_j if there is any vector of primary input values that causes the FSM to evolve from state v_i to state v_j. $W(E)$ is a set of labels attached to each edge. For the purposes of this paper, we define each label as an ordered 4-tuple $< i, s, s', o >$ where i is a minterm over the primary inputs, s and s' are minterms over the state variables and o is a minterm over the primary outputs. The pair $< s', o >$ corresponds to a minterm in the output plane of a truth-table representation of the FSM: for each edge we will refer to the set of all such pairs as the **output-labels** of that edge. This label carries the information of the value of the outputs and next-state resulting from the transition. The pair $< i, s >$ corresponds to a minterm in the input plane of a truth-table representation of the FSM: for each edge we will refer to the set of all such pairs as the **input-labels** of that edge. This label carries the information of the value of the inputs and previous-state that caused the transition. [1]

We denote the primary input combination and present state corresponding to an edge or set of edges as $i \circledcirc s$, where i and s are cubes over the set of inputs and states respectively. The fanin of a state, q is a set of edges and is denoted $fanin(q)$. The fanout of a state q is denoted $fanout(q)$. The output and the fanout state of an edge $(i \circledcirc s) \in E$ are $o((i \circledcirc s))$ and $n((i \circledcirc s)) \in V$ respectively.

A starting or initial state is assumed to exist for a machine, also called the **reset state**. Given a logic-level finite state machine with N_k latches, 2^{N_k} possible states exist in the machine. A state which can be reached from the reset state via some input vector sequence is called a **valid state** in the STG. The input vector sequence is called the **justification sequence** for that state. A state for which no justification sequence exists is called an **invalid state**. Given a fault F, the STG of the machine with the fault is denoted G^F. A **differentiating sequence** for states s_1 and s_2 in a machine is a sequence of inputs $i_1 \ldots i_N$ such that if the machine begins in state s_1, the output associated with input i_N is different than if the machine begins in state s_2. Two states in a STG G are **equivalent** if they do not have a differentiating sequence.

A STG G_1 is said to be **isomorphic** to another STG G_2 if and only if they are identical except for a renaming of states.

The fault model assumed is **single stuck-at**. A finite state machine is assumed to be implemented by combinational logic and feedback registers. Tests are generated for stuck-at faults in the combinational logic part.

A primitive gate in a network is **prime** if none of its inputs can be removed without causing the resulting circuit to be functionally different. A gate is **irredundant** if its removal causes the resulting circuit to be functionally different. A gate-level circuit is said to be **prime** if all the gates are prime and **irredundant** if all the gates are irredundant. It can be shown that a gate-level circuit is prime and irredundant if and only if it is 100% testable for all single stuck-at faults.

We differentiate between two kinds of redundancies in a sequential circuit. If the effect of the fault cannot be observed at the primary outputs or the next state lines, beginning from any state, with any input vector, the fault is deemed **combinationally redundant**. A **sequentially redundant** fault is a fault that cannot be detected by any input sequence and is not combinationally redundant.

To detect a fault in a sequential machine, the machine has to be placed in a state which can then excite and propagate the effect of the fault to the primary outputs. The first step of reaching the state in question is called **state justification**. The second step is called **fault excitation-and-propagation**.

An edge in a STG of a machine is said to be **corrupted** by a fault if either the fanout state or an output-label of this edge is changed because of the existence of the fault. A path in a STG is said to be corrupted if at least one edge in the path has been corrupted.

Internal single stuck-at faults in a logic network are faults on internal lines (not primary input or primary outputs) that are not equivalent to single or multiple primary output stuck-at faults.

3 Redundancies in Sequential Circuits

In this section we characterize redundancies in sequential machines. We present two views of the redundancies by looking at the effect of a fault on the faulty State Transition Graph as well as on the gate-level implementation of the machine.

[1] The reader need not be concerned over this rather verbose description of an FSM: it is used only for notational convenience in the proofs and none of the algorithms require it

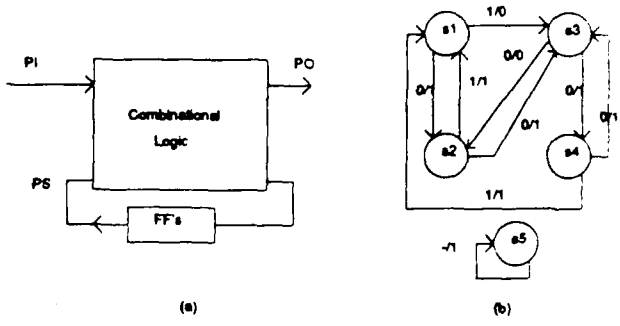


Figure 1: A Sequential Circuit

A general model of a sequential circuit S, implementing a single FSM is shown in Figure 1(a). Gates in the combinational network may be in the cone of the output logic, the next-state logic, or both. The State Transition Graph corresponding to one such machine is shown in Figure 1(b).

Redundant faults in S may be combinationally redundant (CRFs) or sequentially redundant (SRFs). Combinationally redundant CRFs can be eliminated via combinational logic optimization alone [2] [1] [6] and will not be discussed here. Sequentially redundant faults can be classified into three categories [8].

1. *invalid-state faults:* The fault does not corrupt any fanout edge of a valid state in the STG, but does corrupt the fanout edge of an invalid state.

2. *isomorphic faults:* The fault results in a faulty machine that is isomorphic (with a different encoding) to the original machine.

3. *equivalent-state faults:* The fault causes the interchange/creation of equivalent states in the STG.

In [8], it was shown that any sequential redundancy must fall into one of these classifications.

Let us now look at some examples of these faults. A faulty STG corresponding to an *invalid-state SRF* is shown in Figure 2(b). Only fanout edges from an invalid-state have been corrupted. This corresponds to either output/next-state logic that is not combinationally redundant but requires for detection that the state register of the machine be filled with a state code that does not correspond to any valid state. These redundancies actually do occur in practice when the next-state logic has been optimized independently of the state assignment.

The effects of an *isomorphism SRF* are shown in Figure 2(c), where an isomorphic faulty machine (equivalent to the true machine) is depicted in which $s2$ and $s3$ have been interchanged. This occurs when the next-state logic in the good machine which produced the state code associated with $s2$, now produces the state code for $s3$, and *vice versa*. Furthermore, the output logic is simultaneously modified by the fault in such a way that the outputs due to state codes $s2$ and $s3$ are also swapped.

In Figure 1(b), note that states $s2$ and $s4$ are equivalent states. An *equivalent-state SRF* in S may produce the faulty STG of Figure 2(a), where the only input-label associated with the edge $< s1, s2 >$ is moved to a new edge $< s1, s4 >$. Furthermore, the fault does not change the terminal behavior of S. As $s2$ and $s4$ are equivalent, the fault is undetectable. This corresponds to a logic level change such that when the state register holds the code for $s1$, on the input "0" the faulty next-state logic produces the state code for $s4$ rather than the state code for $s2$.

Creating an irredundant sequential machine entails eliminating the sources of redundant faults. In the next section we describe some general procedures which eliminate the isomorphism SRFs and invalid-state SRFs, and partially eliminate equivalent-state SRFs.

4 Eliminating Redundancies in Sequential Circuits

In this section, we will discuss general methods for eliminating certain classes of redundancies in sequential circuits. We will show that simple procedures may eliminate invalid-state and isomorphism SRFs, but the difficulty in synthesizing fully testable sequential machines is in eliminating equivalent-state SRFs. [2] First, we give two results that relate to the elimination of all three classes of SRFs.

[2] It may be worth noting here that despite the apparently greater complexity of sequential test generation relative to combinational test generation, this problem is

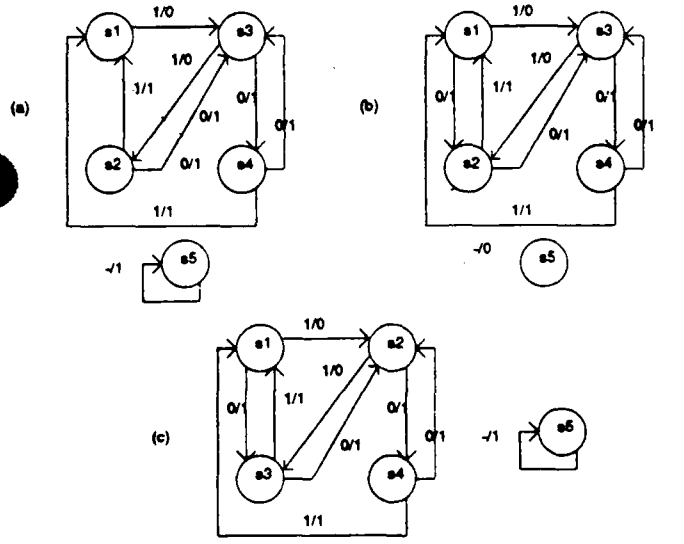


Figure 2: 3 Types of Sequential Redundancies

4.1 Theorems Regarding Unconditional Testability

Variations of the results below were proven in [8] (*c.f.* Lemma 4.1, Theorems 4.2 and 4.4).

Lemma 4.1 : *Given a reduced sequential machine (implemented as in Figure 1) with $N_s \leq 2^n$ states, where n is the number of latches in the machine, all single stuck-at faults on the primary input (PI)/present state (PS) lines and all single and multiple stuck-at faults on primary output (PO)/next-state lines (NS) are testable, if the combinational logic of the machine is prime and irredundant with respect to the invalid state don't care set.*

This lemma usefully allows us to limit our consideration of faults for any machine, as long as we have made the combinational logic prime and irredundant.

Theorem 4.1 : *Given a reduced sequential machine with 2^n states, where n is the number of latches in the machine, if the combinational logic of the machine is prime and irredundant and is implemented in two-level form or algebraically factored multi-level form, then the machine is fully testable for all single stuck-at faults in the combinational logic.*

Proof: The terminal behavior of a reduced machine with 2^n states can only be realized by a machine with $\geq 2^n$ states. No fault in the machine can increase the number of states in the machine. Therefore, the number of states in the faulty STG G^F for any fault F is less than or equal to 2^n. If $|G^F| < 2^n$, then F is testable, since G^F cannot realize the terminal behavior of the true STG G. If $|G^F| = 2^n$, then G^F has to be isomorphic to G in order to realize the terminal behavior of G.

Isomorphism implies an interchange of states and associated edges in the STG of a machine. By Lemma 4.1, we only have to consider internal stuck-at faults in the two-level or algebraically factored multi-level network. If for each internal fault, the parity of inversions is the same (either odd or even) for all paths to the next-state latches, then isomorphism SRFs will not occur. If this inversion-parity invariant is maintained then all the input-labels corrupted by a single internal-fault uniformly result in all state codes in the faulty machine monotonically increasing or monotonically decreasing but not both. Thus, a single fault could not lead to the swapping of state codes required to produce an isomorphism SRF. For example, a s-a-0 fault might result in the next-state logic in the faulty machine producing state code < 001 > rather than < 101 >, but the same fault could not also cause the next-state logic in the faulty machine to produce state code < 101 > rather than < 001 >. The inversion-parity invariance is naturally produced by a number of current synthesis procedures. Clearly internal faults in a two-level combinational network are inversion-parity invariant because all inverters are on the primary inputs. Similarly, circuits obtained via algebraic factorization from two-level networks may also be directly expressed such that all their inverters are on the primary inputs. **Q.E.D.**

The above theorem does not hold for FSMs implemented by general multi-level networks, nor for FSMs with State Transition Graphs (STGs) with fewer than 2^n states, where n is the number of latches in the machine. In Section 5 we define the notion of fault-effect disjointness which, when applied in a synthesis procedure, can guarantee the complete testability of a general sequential circuit by ensuring that each faulty state has an uncorruptible differentiating sequence. We now proceed to discuss techniques for the elimination of particular SRFs.

4.2 Eliminating Invalid-state SRFs

To eliminate these SRFs, it is sufficient to use codes corresponding to invalid states as don't cares during logic optimization. An invalid-state SRF is due to the sub-optimal usage (or no usage) of these don't cares. These redundancies will not exist if the combinational logic is made irredundant under this don't care set.

4.3 Eliminating Isomorphism SRFs

There are many ways of ensuring that isomorphism does not occur due to faults in sequential circuits. Isomorphism due to a fault is essentially due to a *sub-optimal state assignment*. The new encoding corresponding to the isomorph represents a better machine (one with the redundant line removed). *A locally optimal state assignment across any given* set of states can ensure that isomorphism does not occur in multi-level circuits, across this set of states. It is worthwhile to note that optimal state assignment corresponds to the optimal usage of don't cares – one does not care what the codes of the different states are so long as they are distinct.

Two-level realizations and algebraic factorization also eliminate the possibility of isomorphism SRFs (by the arguments used in Theorem 4.1).

4.4 Eliminating Equivalent-state SRFs

Equivalent-state SRFs are related to equivalent valid/valid and valid/invalid state pairs a sequential machine. Given a reduced machine, a fault that corrupts a single edge going to a faulty, but valid, state can not be responsible for a SRF, since all valid states are distinguishable. Thus, an initial state minimization will preclude the occurrence of the SRF of the form in Figure 2(a). However, we may have a case where the fault results in a invalid next state that is equivalent (or becomes equivalent) to the true next state. This is illustrated in Figure 3. We have the true STG in Figure 3(a), that is state minimal. The invalid state $s4$'s code has been used as a don't care and $s4$ is equivalent to state $s2$ after logic minimization under this don't care condition. A fault could result in the scenario shown in Figure 3(b), where a single corrupted edge whose true next state is $s2$ produces a faulty next state, $s4$. The fault is redundant. Equivalent-state SRFs due to these valid/invalid state pairs pose major difficulties for testability-driven synthesis and we devote the remainder of the paper to discussing a spectrum of techniques that eliminate them.

5 Distinguishing Sequences and Equivalent-State SRFs

The most general paradigm for the elimination of equivalent-state SRFs is to ensure that for each faulty/fault-free state pair produced by a fault, at least one differentiating sequences exists which is not destroyed by that fault. This is a necessary and sufficient condition and while obvious, we encapsulate it in the following observation.

Theorem 5.1 : *Given a sequential machine with no combinationally redundant faults, invalid-state SRFs or isomorphism SRFs, if for each fault in the machine at least one (possibly multiple-vector) differentiating sequence for at least one faulty/fault-free state pair produced by a fault in the machine is retained in spite of the fault, then the resulting sequential machine is fully testable.*

There are two conditions under which a differentiating sequence is retained. The first is that the fault which produces the faulty/fault-free pair does in fact corrupt the differentiating sequence, but the behavior of the faulty machine is still distinguishable from the good machine. For

also NP-complete when the input circuit is accompanied by a FSM description. The sequential test generation problem is clearly NP-hard, and as any test sequence for a fault is bounded by the size of the input FSM, a test sequence may be verified in polynomial time by a fault simulator. Thus the sequential test generation problem is also contained in NP.

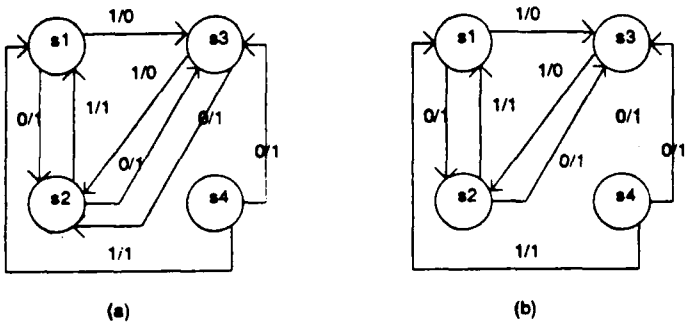


Figure 3: A Complex Equivalent State SRF

instance, say two states q_1 and q_2 on receiving the input i_1 produce outputs o_1 and o_2 respectively. if q_1/q_2 appears as a faulty/fault-free state pair due to a fault, f, then f may corrupt o_1 to $o_1{}'$. The differentiating input i_1 is still retained if $o_1{}' \neq o_2$. This condition is discussed in Section 5.4. The second condition for retainment is that f does not corrupt o_1, or more generally that any fault which creates an faulty/fault-free state pair does not corrupt the differentiating sequence for that state pair. We discuss this in the following sections.

5.1 Fault-Effect Disjointness

The elimination of equivalent-state SRFs is ensured when differentiating sequences, for possible faulty/fault-free state pairs produced due to a fault, are uncorrupted by that fault. These sequences may, of course, be corrupted by other faults. This is accomplished by defining the notion of fault-effect disjointness (FE-disjointness) between a pair of edges and applying it to combinational networks.

Definition 5.1 : *Given a FSM M, a STG G representing M and a logic-level implementation L of M, a fault f is said to perturb an input-label m of an edge e in G iff the fault in L causes the input-label to be removed from e (and moved to another edge).*

Every fault that perturbs an edge corrupts the edge, but a fault may corrupt an edge without changing the fanout state; whereas every fault that perturbs an edge changes the fanout state.

Definition 5.2 : *Given a FSM M and a STG G representing M, a logic-level implementation L of M, and two input-labels m_1 and m_2 of two edges e_1 and e_2 in G, the two labels m_1 and m_2 are said to be FE-disjoint over a set of faults $F \in L$ if no fault in F corrupts both m_1 and m_2.*

Based on FE-disjointness alone, we can define a general procedure that produces fully testable sequential machines.

Theorem 5.2 : *If each of the input-labels in at least one (possibly multiple-vector) differentiating sequence of at least one faulty/ fault-free state pair produced by a fault in the machine are made FE-disjoint from the input-label whose perturbation caused the faulty/fault-free state pair, then the resulting sequential machine is fully testable.*

Proof: Since at least one differentiating sequence for a faulty/ fault-free pair that is produced due to a fault is uncorrupted by the fault, traversing the input-labels in the differentiating sequence will detect the fault at the primary outputs. **Q.E.D.**

The following points are worthy of note:

1. **Possible faulty/fault-free pairs**: An extreme case corresponds to a fault resulting in all possible pairs of states becoming faulty/fault-free state pairs. However, depending on the type of implementation, the effect of a fault varies. For example, internal faults in a two-level or algebraically factored network uniformly produce a 0 instead of a 1, or a 1 instead of a 0 at the outputs they are propagated to. Logic partitioning can restrict the set of outputs a fault can be propagated to, in two-level or general multi-level networks. Synthesis procedures can be characterized by restrictions on faulty/fault-free state pairs that can occur, placed via constraints on logic optimization.

2. **State assignment**: State encoding controls what *symbolic* states are produced as faulty/fault-free pairs. Constrained state assignment can be used in conjunction with logic optimization to restrict what symbolic states can appear as faulty/fault-free state pairs. We do not explore this approach further in this paper.

3. **Obtaining FE-disjointness**: For any fault f, a valid/invalid state pair is first activated by an input in a particular state, *i.e.*, by an input-label. Each of our procedures ensures that the fault f which perturbs the input-label m_1 and produces the valid/invalid state pair does not also corrupt the differentiating sequence (*e.g.* m_2) for the invalid/valid state pair. This is ensured by making the input labels m_1 and m_2 FE-disjoint. There are several methods of obtaining FE-disjointness for a pair of input-labels over a fault in a FSM implemented by two-level or multi-level combinational logic; different methods are characteristic of different synthesis approaches. For example, partitioning the output and next state logic in a sequential machine ensures that the output of a faulty state (produced by a fault in the NSL block) is not corrupted by the fault. Optimal usage of don't cares represents another technique to ensure FE-disjointness.

4. **Multiple-cycle differentiating sequences**: In general, differentiating sequences for a given pair of states may have lengths greater than 1. In this case, we require the input-label m_1 which activated the faulty/fault-free state pair to be FE-disjoint from each of the input-labels $m_2 \ldots m_N$ corresponding to the differentiating sequence for the faulty state.

We will now show how previously proposed synthesis procedures can be viewed as different approaches to insuring the invariant given in Theorem 5.2. In Section 5.2 we consider procedures that ensure FE-disjointness through a highly restricted implementation. This procedure has the advantages that it is computationally inexpensive and the time for generating tests for the resulting logic is also reduced (see Section 6).

In Section 5.3 we show that FE-disjointness can be maintained in two-level circuits by modifying the initial Boolean cover. The FE-disjointness invariant can then be further retained in a multi-level implementation by constraining the algebraic factorization of the two-level implementation. The resulting implementation has significantly fewer restrictions than the implementations resulting from the constrained synthesis procedures described in Section 5.2 and this results in smaller implementations (see Section 6). The procedures based on covering and factorization also have many degrees of freedom in their application. They may be applied so as to minimize computation time with the potential for an inferior implementation or they may be applied so as minimize the size of the implementation at greater computational cost.

Finally, in Section 5.5 we discuss a procedure that achieves an optimal implementation by iteratively removing SRFs. Such an approach maintains the FE-disjointness invariant as well. From the results in Section 6 we see that this approach is the most computationally expensive but also produces the smallest logic.

5.2 Constrained Synthesis Procedures

The procedure of [7] adds edges to the initial STG specification to raise the number of states in the STG to 2^n, where n is the number of latches in the machines. Thus, no invalid states exist in the machine. If the added pseudo-valid states are not equivalent to the other states in the machine, then by Theorem 4.1, full testability is obtained in a two-level or algebraically factored multi-level implementation. The procedure ensures full *and* easy testability in a general multi-level logic implementation, via constrained state assignment and logic partitioning. [3] The synthesized machine is easily testable in the sense that the length of a differentiating sequence for any possible faulty/fault-free state pair is limited to 1.

In Figure 4, the architecture used by the procedure for a Mealy machine is shown. Each of the next state (NS) lines has been realized as a separate circuit. The constraint on the state assignment is that any pair of states that cannot be distinguished via a single-vector sequence be given codes at least of distance-2. We state the following theorem to put the procedure of [7] in context of FE-disjointness and Theorem 5.2. Only internal faults are considered since PI/PS/NS/PO faults are testable by Lemma 4.1.

[3] In this procedure a locally optimal state assignment is not required for full testability in a general multi-level implementation.

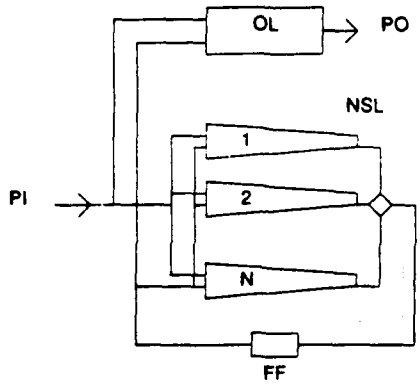


Figure 4: Architecture of Partitioned Mealy Machine

Theorem 5.3 : *The procedure of [7] results in a machine where all possible faulty/fault-free state pairs due to an internal fault have differentiating sequences of length 1, that are FE-disjoint from the input-label whose perturbation caused the faulty state.*

Proof: An internal fault in the OL block can only be propagated to the POs and thus cannot cause a faulty state. Without loss of generality, consider a fault in 1st NS line partition. Since the combinational logic is irredundant, an input-label and present state exist that propagate the effect of the fault to the 1st NS line. Therefore, the faulty state will differ from the true state in the 1st bit alone. The state encoding is such that the faulty/fault-free state pair possesses a differentiating sequence of length 1. The partitioning of the OL and NSL blocks guarantees FE-disjointness of the differentiating vector from the input-label whose perturbation caused the faulty/fault-free state pair. **Q.E.D.**

Note that this theorem in conjunction with Theorem 5.2 ensures full/easy testability for the synthesized FSM.

The constrained synthesis procedure that we presented here maintains fault-effect disjointness at a considerable area penalty. In the following sections we present procedures that are less restrictive on the optimization steps in synthesis.

5.3 Retaining FE-Disjointness Through Covering and Factorization

5.3.1 Fully Testable Machines with Two-level Logic Implementations

The notion of fault-effect disjointness (FE-disjointness) can be applied to two-level logic minimization in order to produce two-level combinational logic networks implementing FSMs that are fully testable. The procedure described here is primarily concerned with differentiating sequences of faulty fault-free state pairs. These pairs are such that the faulty state is an invalid state, since, by the arguments of Theorem 4.1, if only valid states are produced as faulty states, full testability can be obtained via a standard minimization strategy. Also, we will be dealing only with internal faults in the network; Lemma 4.1 guarantees the unconditional testability of primary input, present state line, next state line and primary output stuck-at faults.

The strategy used here modifies the logic minimization process using the invalid states as don't cares, so for each invalid state w the following conditions are satisfied.

1. *If* is not required to detect any fault F in the machine S.
2. *If* is distinguishable from any valid state in a specified number (≥ 1) of state transitions or *it* never appears as a faulty next state, that is equivalent to the true next state.

The goal of the minimization procedure is to satisfy Conditions 1 and 2 *and* produce an area-minimal logic circuit. The prime implicant generation and covering steps that are basic to two-level Boolean minimization are modified to this end.

We now apply the notion of FE-disjointness to two-level networks.

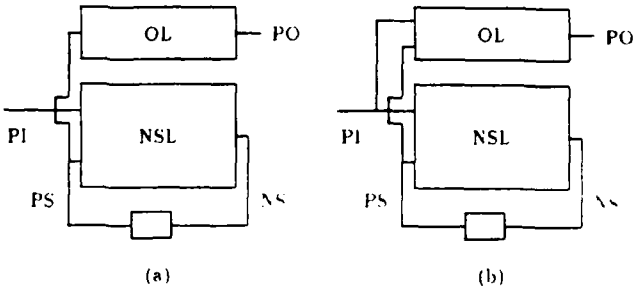


Figure 5: Moore and Mealy Finite State Machines

Definition 5.3 : *A Distance-k-prime-cube (D-k-prime-cube) of a prime cube c is a cube that has exactly the variables of c and a 1 (0) in exactly k positions where c has a 0 (1), in any combination.*

It is only meaningful to talk about a D-k-prime-cube relative to a particular prime cube, but whenever the prime cube that is being referred to is unambiguous we will use the term *D-k-prime-cube* to abbreviate *D-k-prime-cube relative to a prime cube.*

Lemma 5.1 : *Given M, G and a two-level implementation of T of M, and a single internal fault f in T that perturbs an input-label m of an edge e in G, if f is a s-a-0 fault on the output of an AND gate g of T then m is contained within the prime cube associated with g, and if f is a s-a-1 fault on the input of an AND gate g_j of T then m is contained within a D-1-prime-cube relative to g_j.*

Proof: First, observe that we can collapse the internal faults in a two-level network to s-a-1 faults on the AND gate inputs, s-a-0 faults at AND gate outputs and s-a-0 faults at OR gate inputs. S-a-1 faults at AND gate outputs and OR gate inputs are equivalent to single or multiple PO s-a-1 faults. S-a-0 faults at AND gate inputs are equivalent to the corresponding s-a-0 fault at the AND gate output.

Suppose f is a s-a-0 fault at the output of an AND gate g. To perturb m, f must cause m to move to another edge e_k. Only those input-labels contained within the prime cube associated with g, will be affected by the s-a-0 fault, thus m is contained within the prime cube associated with g_i. Note that assuming complete observability of all outputs and next-state lines one can view m as a test vector for f.

By a similar argument, for an input label m to be affected by a s-a-0 fault f on an OR gate input, m must be contained within the prime cube associated with the gate g_i that fans out to the affected OR gate. Thus the set of input labels perturbed by a s-a-0 fault on an OR gate input that is fed by an AND gate g_i will always be a subset of the input labels affected by a s-a-0 fault on the input (or output) of g_i.

Suppose f is a s-a-1 fault at the input of an AND gate g_j. To perturb m, f must cause m to move to another edge e_k. Only those input-labels contained within the D-1-prime-cube relative to the prime cube associated with g_j will be affected by the s-a-1 fault, thus m is contained within the D-1-prime-cube. As before, assuming complete observability of all outputs and next-state lines one can view m as a test vector for f. **Q.E.D.**

We can state a theorem regarding sufficient conditions for two edge labels to be FE-disjoint over s-a-0 or s-a-1 internal faults in a two-level network.

Theorem 5.4 : *Given M, G and T as above, two input-labels m_1 and m_2 are FE-disjoint over internal s-a-0 (s-a-1) faults in a two-level network, if one of the following conditions is satisfied:*

1. *m_1 and m_2 are not both contained in any prime (not both contained in any D-1-prime-cube) in T.*
2. *m_1 and m_2 are both contained in a prime p_1 (or in a D-1-prime-cube of a prime p_2), and m_1 or m_2 is contained in some other prime p_3 that asserts the same outputs as the prime p_1 (or p_2).*

Proof: First, consider Condition 1. By Lemma 5.1, for an input-label to be perturbed by a s-a-0 fault it must be contained in the prime cube associated with the faulty gate. Similarly, for an input-label to be perturbed by a s-a-1 fault it must be contained in the D-1-prime-cube associated with the faulty gate. Thus for two input-labels to both be

corrupted by a given s-a-0 (s-a-1) fault they must both be contained within the same prime cube (D-1-prime-cube). If this condition is not met then no fault can simultaneously perturb both edge labels.

Next, consider Condition 2. If F is at the input l of an AND gate, for F to truly perturb an input-label m no other AND gate feeding the OR gate asserting the PO, can have a 1 at its output on m. This implies that if m is perturbed by F, for each PO fed by g_1, the input-label m cannot be contained in any of the primes corresponding to the other AND gates feeding the PO. Thus, the input-labels perturbed by F are restricted to those that serve as primality tests for l in p_1. Consider a s-a-0 fault, F, at the output of an AND gate g_2 and associated prime p_2. For F to perturb an input-label m, no AND gate feeding the same OR gate(s) as g_2 can have a 1 at its output on m. This implies that if m is contained in a prime asserting the same outputs as g_2, it will not be perturbed by F. Thus, the input-labels perturbed by F are restricted to those that serve as irredundancy tests for p_2. **Q.E.D.**

We now define a procedure that produces a fully testable Moore machine, under the architecture of Figure 5(a).

1. The OL block is minimized with the invalid states used as don't cares, attempting to make sure that a maximal number of invalid states produce different output combinations from all or a maximal number of valid states. If all invalid states produce different outputs from each of the valid states, unconditionally minimize the NSL block and exit. (Two invalid states are allowed to produce the same output).

2. For each invalid state ir_k, find the set of valid states $Q_k = q_{k1}, .. q_{kN_k}$, that assert the same output combination as the invalid state, and such that $ir_k \supset q_{kj}$ or $q_{kj} \supset ir_k$.

3. Perform a two-level Boolean minimization on the logic of the NSL block, meeting the following conditions:

   (a) Use the invalid states as don't cares for all primary input values.

   (b) For each invalid state ir_k, ensure that there exists a PI vector i_{kj} that distinguishes ir_k and $q_{kj} \in Q_k$, $1 \leq j \leq N_k$. That is, i_{kj} produces different next states for ir_k and q_{kj}, such that the next states assert different output combinations, via an appropriate selection of primes. Also, the vector pairs corresponding to $v \in fanin(q_{kj})$ and $i_{kj} \oplus ir_k$ are constrained to be FE-disjoint over (each individual fault in) the s-a-0 (s-a-1) internal faults in the network corresponding to the cover if $q_{kj} \supset ir_k$ ($ir_k \supset q_{kj}$), via an appropriate selection of primes that satisfy the conditions of Theorem 5.4.

Theorem 5.5 : *If the procedure above completes successfully, it produces a fully testable Moore machine.*

Proof: Faults in the OL block can be detected by justification sequences to the appropriate valid states that propagate the fault to the POs.

Consider an internal fault F in the NSL block. If F results only in faulty next states that are valid states or invalid states asserting different output combinations from the true valid state, then F is testable. We have to consider the possibility of F resulting in a faulty/fault-free state pair that corresponds to an invalid-valid state pair, namely ir_k, q_{kj}, which both assert the same output combination.

Since F is an internal fault, it can only monotonically increase the faulty state bits or monotonically decrease them (*c.f.* Theorem 4.1). Therefore, $ir_k \supset q_{kj}$ or $q_{kj} \supset ir_k$. We can thus discard faulty/fault-free state pairs that do not satisfy these conditions at Step 2. If ir_k, q_{kj} appeared as a faulty/fault-free pair, it means that $v \in fanin(q_{kj})$ was corrupted to ir_k, instead of q_{kj}. If $q_{kj} \supset ir_k$, then it means we are dealing with a s-a-0 fault. Then, a differentiating vector i_{kj} for ir_k, q_{kj} will not have been corrupted since $i_{kj} \oplus ir_k$ and v are FE-disjoint over the s-a-0 internal fault set. We can similarly argue the s-a-1 case. Thus, we can detect F in the next state transition, via the uncorrupted differentiating vector for ir_k, q_{kj}. **Q.E.D.**

The procedure is easily extended to the Mealy machine case (Figure 5(b)). The procedure to produce a fully testable Mealy machine is similar to the Moore machine procedure, except that during the minimization of the OL block, we can make choices as to what vectors can be used to distinguish the invalid and valid states, while maintaining primality and irredundancy of the OL block cover. During the minimization of the NSL block, we effectively ensure for state pairs that do not have a differentiating vector that a two-vector differentiating sequence for the pair is uncorrupted, if the two states are produced as a faulty/fault-free pair.

Finally, the procedure can be extended to synthesize Moore or Mealy machines under the lumped architecture of Figure 1(a). In this case, we have more FE-disjointness constraints, since we have to ensure that the output asserted by an invalid state (under some primary input combination) is uncorrupted if the state is produced as a faulty state. If the output is not distinct from the output produced by the true state, then the next state of the faulty state has to satisfy the condition described in Step 3(b) above.

5.3.2 Fully Testable Machines with Multi-level Logic Implementations

We wish to extend the results of the previous section to multi-level implementations. As before the paradigm followed is to ensure that the differentiating sequences, for possible faulty/fault-free state pairs produced due to a fault, are uncorrupted by that fault. This is accomplished by applying the notion of FE-disjointness between a pair of edges to multi-level combinational networks. Guaranteeing FE-disjointness between two input-labels is more complicated in a multi-level implementation than in a two-level implementation. This is due to the fact that a single fault in a multi-level implementation may be equivalent to a multiple fault in a two-level network. To simplify things we restrict our consideration to those multi-level networks that are the result of an algebraic factorization [4] of a prime and irredundant two-level network. Unfortunately, space limitations make a review of key logic synthesis concepts such as cube, kernel, kernel-cube and factor impossible, but [3] gives a good treatment of these ideas. Recently, it was shown in [9] that each single internal fault in an multi-level implementation that was algebraically factored from a prime and irredundant two-level network is equivalent to a multiple internal fault in the two-level network. In particular, it can be shown that any single internal s-a-0 (s-a-1) fault in an algebraically factored network is equivalent to a s-a-0 (s-a-1) multiple fault in the associated two-level network. We therefore begin with perturbation conditions for input-labels under a multiple fault in two-level networks, and then apply these results to algebraically factored networks.

Lemma 5.2 : *Given M, G and T as in Lemma 5.1 and a multiple s-a-0 internal fault f in T, if f perturbs an input-label m in G then every prime in which m is contained must have been affected by the fault. Furthermore, that perturbation causes some next-state variable that formerly was 1 to become 0.*

Proof: The effect of an internal s-a-0 fault on a prime is to remove that prime from the cover. The effect of a number of internal s-a-0 faults is to remove each affected prime from the cover. These missing primes affect the network in a predictable way: If all the primes that covered an input-label are missing then that input-label which formerly resulted in some next-state or primary output variables having the value 1 now results in those same variables having the value 0. If next-state variables are affected then the input-label is perturbed. Note that it is necessary for all primes covering an input-label to be affected before the input-label perturbed. **Q.E.D.**

We wish to use this lemma to arrive at conditions for input-labels to remain FE-disjoint in the presence of a single internal s-a-0 fault in an algebraically factored multi-level network.

Theorem 5.6 : *Given M, G and T as above, let A be an algebraic factorization of T. Let m_1 and m_2 be two input-labels of G and let P_1 be the set of all primes of T that cover m_1 and let P_2 be the set of all primes of T that cover m_2. The two input-labels m_1 and m_2 in G are FE-disjoint over internal s-a-0 faults in A, if both m_1 and m_2 are not contained in any single prime cube in T and no factor extracted in the factorization of A contains cubes common to every prime in P_1 and every prime in P_2.*

Proof: That both m_1 and m_2 are not contained in any single prime cube in T is simply restating the condition of Theorem 5.4. Note that this condition implies that P_1 and P_2 are disjoint. By Lemma 5.2, in order for a s-a-0 fault to perturb m_1, it must affect every prime in P_1 and similarly in order for a s-a-0 fault to perturb m_2 it must affect every prime in P_2. [4] If a single s-a-0 internal fault in A perturbs m_1 and m_2 when the fault is applied and A is collapsed to two-levels (the inverse operation of factorization), then every prime in both P_1 and P_2 must have been affected. For this to occur, during factorization there must either be some cube factor c such that c is a sub-cube of every prime in both P_1 and P_2, or there must be some kernel factor k such that some kernel-cube of k is a sub-cube of every prime in P_1 and every prime in P_2. **Q.E.D.**

Using these results to arrive at an algebraic factorization A in which m_1 and m_2 are FE-disjoint with respect to any internal s-a-0 fault requires first building sets P_1 and P_2. During cube extraction, a cube

[4] Certainly, primary input faults can produce such an effect but such faults are easily detectable by other means.

c is eliminated from consideration as a factor if c is a sub-cube of every prime in P_1 and every prime in P_2. If this cube were allowed, a s-a-0 fault on the output of the gate associated with the cube could potentially eliminate all primes in P_1 and P_2 and as a result perturb both m_1 and m_2. During kernel extraction [3], a kernel k is eliminated from consideration as a factor if every prime in P_1 and every prime in P_2 contains as a sub-cube some kernel-cube (it need not be the same kernel-cube in each case) of k. If this kernel were allowed, a s-a-0 fault on the output of the gate associated with the kernel could potentially eliminate all primes in P_1 and P_2 and as a result perturb both m_1 and m_2. All other factors are viable. It is worth noting that while such factors that violate the FE-disjointness condition may exist, they appear to be highly unlikely.

Characterizing the influence of s-a-1 faults in a multi-level network is more complicated than that regarding s-a-0 faults. In the case of s-a-0 faults, each input-label m that is a member of the ON-set is covered by some set of primes and for m to be perturbed *all* of those primes must be affected by some s-a-0 fault. If an input-label m is a member of the OFF-set then for each prime p in T there exists a k such that m is contained in a D-k-prime with respect to p, and a multiple s-a-1 fault affecting *any* of the primes in T may perturb m.

Lemma 5.3 : *Given M, G and T as above, and a multiple s-a-1 internal fault f in T, if f perturbs an input-label m in G then m is contained within a D-k-prime relative to an affected prime of T and m is not contained in any other prime of T. Furthermore, that perturbation results in some next-state variable that formerly was 0 to become 1.*

Proof: The effect of an internal s-a-1 fault on a prime is to expand that prime in the cover. The effect of a number of internal s-a-1 faults is to expand each affected prime from the cover in each literal that is s-a-1. These expanded literals affect the network in a predictable way: Some input-labels that formerly resulted in primary outputs and/or next state variables being 0 now result in those same variables being 1. The input-labels that will be thus affected are exactly those input-labels that are contained within a D-k-prime relative to an affected prime. For instance, if the prime cube $abcd$ is affected by faults a and b s-a-1 then any input-label contained in the D-1-primes $a'bcd$ or $ab'cd$ or the D-2-prime $a'b'cd$ will be perturbed by this multiple fault, unless it is already contained within some other prime in T. **Q.E.D.**

We wish to use this lemma to arrive at conditions for input-labels to remain FE-disjoint in the presence of a single internal s-a-1 fault in a algebraically factored multi-level network.

Theorem 5.7 : *Given M, G and T as above, let A be an algebraic factorization of T. Let m_1 and m_2 be two input-labels of G. The two input-labels m_1 and m_2 in G are FE-disjoint over internal s-a-1 faults in A, if no factor of A contains a cube c such that if each literal of c is expanded in each prime in A in which c appears, then there does not exist an expanded prime p in T that covers m_1 and an expanded prime q in T that covers m_2.*

Proof: We are concerned with identifying the circumstances under which both m_1 and m_2 are perturbed by a single s-a-1 fault. For each of m_1 and m_2 to be perturbed, it must be contained in some expanded cube. A s-a-1 fault in a cube factor c results in raising each literal of c in each prime in T from which c was factored. If no expansion resulting from a s-a-1 fault on any factored cube simultaneously covers m_1 and m_2, then m_1 and m_2 are FE-disjoint under any internal s-a-1 fault. **Q.E.D.**

To use these results to arrive at an algebraic factorization A in which m_1 and m_2 are FE-disjoint with respect to any internal s-a-1 fault, it is sufficient to consider the impact on the network of a s-a-1 fault on each potential factor. Specifically, during kernel extraction, a kernel-cube c is eliminated from consideration as a factor, if expanding the literals of c in each prime in T in which c appears, results in an expanded prime p that covers m_1 and an expanded prime q that covers m_2. Similarly, in cube extraction, a cube c is eliminated from consideration as a factor, if expanding the literals of c in each prime in T in which c appears, results in an expanded prime p that covers m_1 and an expanded prime q that covers m_2. For example, assume we are given the function $F = abcd' + abc'd$ and input-labels $m_1 = a'bcd'$ and $m_2 = ab'c'd$. The cube ab would not be considered as a factor because a s-a-1 fault on ab would result in m_1 being perturbed by the expansion of the prime $abcd'$ to cd' and m_2 being perturbed by the expansion of prime $abc'd$ to $c'd$.

5.4 Fault Simulation

The procedures discussed in Section 5.1 seek to retain differentiating sequences by ensuring that any fault which produces the faulty/fault-free state pair cannot corrupt the the differentiating sequence for that pair in any way. In this section we consider the situation in which the fault that produces the faulty/fault-free state pair does in fact corrupt the differentiating sequence, but the differentiating sequence, or a subsequence of it, still has differentiating behavior for the faulty/fault-free state pair. The circumstances under which this condition occurs are so difficult to classify that we can find no general condition in synthesis which ensures this. [5] For this reason we suggest fault simulation as the best way to recognize the maintenance of a differentiating sequence even when it is corrupted by the fault it was intended to detect.

To motivate this situation, consider the scenario in which we are given a *circuit implementing a sequential machine and for each fault f* in the circuit that produces a non-empty set of faulty/fault-free pairs $P = \{p_1, p_2, \ldots p_m\}$ and for each p_i in P we are given a non-empty set of differentiating sequences $S_i = \{s_{i1}, s_{i2}, \ldots s_{in}\}$ that presently detect the fault. The implementation could have been produced by one of the previous synthesis procedures in the section or by manual design. Similarly, the differentiating sequences could have been produced manually or via automatic test-pattern generation. We then wish to optimize the circuit in such a way that the full testability of the circuit is retained. To be certain that we have a fully testable machine we wish to ensure that these sequences are retained after optimization.

A simple approach to determine if the optimized machine is still fully testable is to fault simulate each s_{ij} on f. If the behavior of any s_{ij} is unchanged by f then a differentiating sequence for f clearly exists. The more interesting case is where the behavior of *all* members of S_i is changed. In this case we further analyze the results of the fault simulation of each s_{ij} to see if the behavior of the faulty machine is still different from the behavior of the true machine. As long as the behavior differs, s_{ij} is still a differentiating sequence for f.

As stated in Theorem 5.1, we only require *one* differentiating sequence for *one* faulty/fault-free state pair produced by a fault to be retained. A fault may produce several faulty/fault-free state pairs based on what excitation vectors are applied. These state pairs will typically have multiple differentiating sequences. A CPU intensive but less restrictive procedure might simulate all possible differentiating sequences for all possible faulty/fault-free state pairs due to a given fault, and check to see if any one of them is retained . A computationally efficient but less optimal/more restrictive procedure may focus on a particular differentiating sequence for each possible faulty/fault-free state pair and checking just the one for retainment.

As mentioned earlier, the differentiating sequences may be either single or multiple vectors. If they consist of multiple vectors a situation may arise where a fault corrupts some input-label or output-label associated with an intermediate input vector, with the result that a subsequence of the original differentiating sequence is now a differentiating sequence. For example, the sequence $s_1 \ldots s_i \ldots s_n$ might be corrupted in such a way that the sub-sequence $s_1 \ldots s_m$ is now a differentiating sequence.

5.5 Optimal Synthesis Procedures

The procedures of Sections 5.1 and 5.4 sought to directly ensure that particular differentiating sequences are retained. Here we review a synthesis procedure that simply guarantees that a differentiating sequence for a valid/invalid state pair will always exist.

The procedure of [8] uses repeated logic minimization to achieve FE-disjointness between each of the input-labels in a differentiating sequence of invalid/valid (faulty/fault-free) state pairs and the input-label whose perturbation caused the invalid faulty state.

Given an incompletely specified combinational logic function, we can obtain a prime and irredundant implementation of the logic function, in two-level or multi-level form, that has the following properties:

1. An input test vector exists for every single stuck-at fault in the logic network that lies outside the don't care (DC) set and in the ON or OFF-sets.

2. At least one of the output values that differ in the true and faulty circuits, on the application of this input test vector, will not correspond to a don't care output condition.

In the procedure of [8], the approach taken is that the redundancy of Figure 3 exists because we have not exploited the don't care corresponding to the edge (0, $s3$): we can specify $n(0, s3) = (s4, s2)$ and not just $s2$. The following procedure of repeated logic minimization guarantees upon convergence that equivalent-state and invalid-state SRFs don't exist in the resulting machine.

eliminate-equivalent-state/isomorphism-SRFs(S):
{

[5] Specific synthesis procedures can exploit the fact that differentiating sequences may be corrupted and still retained.

```
iter = 1 ;
do {
  if ( iter = 1 ) G = extract-stg( S ) ;
  else G = extract-stg( S'' ) ;
  foreach ( valid state q ∈ G ) {
    Find all valid states (v_1, .. v_m) ≡ q ;
    Find all invalid states (iv_1, .. iv_n) ≡ q ;
    DC_1 : fanin(q) : v_1, .. v_m, iv_1, .. iv_n ) ;

    Find all input-labels i_{q,s} differentiating q and s ≢ q ;
    DC_2 : fanin(q) = (q, s) && n(i_{q,s}, q) = n(i_{q,s}, s)
      o(i_{q,s}, q) = o(i_{q,s}, s) ;
  }
  S' = optimize( S, DC_1, DC_2 ) ;
  IV = extract-invalid-states( S' ) ;
  S'' = optimize( S', DC_IV ) ;
  iter = iter + 1 ;
} while( S ≠ S'' ) ;
}
```

The procedure **optimize()** produces a prime and irredundant two-level or multi-level network under a don't care set. DC_1 corresponds to the don't cares described above. DC_2 is a more complex don't care whose usage ensures that the invalid faulty state does not become equivalent to the true valid state. DC_{IV} corresponds to the don't cares due to invalid state codes.

Theorem 5.8 : *The procedure of [8] guarantees that at least one invalid/valid faulty/fault-free state pair produced due to a fault possesses a differentiating sequence that is FE-disjoint from the input-label whose perturbation caused the faulty state.*

Proof: The procedure has specified don't cares corresponding to the equivalence of invalid/valid state pairs. Given that the combinational logic implementation is prime and irredundant under this don't care set, DC_1, we are guaranteed an input-label perturbation outside DC_1, *i.e.* the faulty state produced by the input-label will not be equivalent to the true state. Further, using the don't care set, DC_2, will ensure that the *fanout* of the invalid faulty state is not corrupted to make the invalid state in the faulty machine equivalent to the true state, *i.e.* a differentiating sequence of the invalid/valid state pair produced will be FE-disjoint from the perturbed input-label. **Q.E.D.**

Note that this theorem in conjunction with Theorem 5.2 ensures full testability for the synthesized sequential machine

6 Results

In this section, we present preliminary experimental results using the synthesis algorithms presented in Section 5.

A standard synthesis procedure was first adopted. The procedure is as follows:

1. State minimization.
2. State assignment (unconstrained).
3. Two-level Boolean minimization using the invalid states as don't cares.
4. Multi-level logic optimization (both algebraic as well as Boolean operations).
5. After synthesis, tests were generated for the circuit using the sequential test generator, STALLION.

Next, we used the synthesis procedure described. The procedure was as follows:

1. Same as Step 1 above.
2. Same as Step 2 above.
3. Two-level Boolean minimization with constrained covering.
4. If each invalid state asserts different outputs from all the valid states, then an unconstrained multi-level logic optimization step was performed. Else, two different options were exercised.
   (a) Strictly algebraic factorization was performed. After factorization, the resulting network was analyzed to check for cube and kernel factors that could potentially cause redundancy. The nodes corresponding to these disallowed factors were *pushed* (collapsed) into their fanins.

EX	#inp	#out	#states	#edges	#lat
ex1	9	9	6	24	3
ex2	2	1	13	57	4
bbara	4	2	7	45	3
bbsse	7	7	13	55	4
s1	8	6	20	110	5
dfile	2	1	24	96	5
planet	7	19	48	118	6
scf	27	54	97	168	8
fs1	8	1	65	581	7
fs2	9	1	130	694	8

Table 1: Statistics of Benchmark Examples

EX	STANDARD			COVER-A			COVER-B		
	lit	fcov	tpg time	lit	fcov	tpg time	lit	fcov	tpg time
ex1	45	97.9	2.4s	50	100	2.0s	48	100	2.0s
ex2	57	98.1	48s	63	100	37s	60	100	40s
bbara	68	100	1.8m	68	100	1.5m	68	100	1.6m
bbsse	124	100	3.2m	124	100	2.6m	124	100	2.7m
s1	143	99.8	5.1m	161	100	3.6m	154	100	4.6m
dfile	171	97.8	5.6m	201	100	3.1m	194	100	3.9m
plan	532	100	2.3m	532	100	2.1m	532	100	2.1m
scf	794	100	82m	794	100	44m	794	100	73m
fs1	432	95.8	26m	451	100	11m	440	99.8	13m
fs2	552	94.9	34m	578	100	19m	561	99.6	22m

   (b) An unconstrained multi-level logic optimization was carried out. Note that in this case, we cannot guarantee 100% testability.
5. Sequential test generation can be performed more efficiently in this case than via STALLION, since we already know all the uncorrupted differentiating sequences for each possible faulty/fault-free state pair. Hence, the propagation step in STALLION is avoided.

In Table 1, we give the statistics of the benchmark examples from the MCNC Logic Synthesis Workshop and industrial sources. The results obtained on these examples via running the standard synthesis procedure and the two options in the new procedure are summarized in Table 2 under the columns STANDARD, COVER-A and COVER-B. The number of literals in the combinational logic (lit), fault coverage obtained (fcov) and the CPU time for test generation (tpg time) are indicated in the three cases. **fs1** and **fs2** are particularly vicious examples. They each have a large number of states and a single output. All the CPU times are on a VAX 11/8800.

COVER-A results in 100% testable designs with small area overheads, that require less CPU time for test generation than the STANDARD procedure. We cannot guarantee full testability via COVER-B, but it allows for the use of more powerful Boolean operations and hence the area overhead is smaller than via COVER-A. Highly (> 99%) testable realizations are obtained in all cases via COVER-B.

We next compare this approach with previously proposed synthesis approaches to achieve full testability. The comparisons are presented in Table 3. Under the column COVER, we give the result corresponding to COVER-B, if the resulting design was fully testable. Else, we give the result of COVER-A. The column CONSTRAIN has the results obtained by using the constrained state assignment and logic optimization procedure of [7]. The column OPTSYN has the results using the optimal synthesis procedure of [8]. The number of literals in the combinational logic (lit), the CPU time for synthesis (syn. time) and the CPU time required for test generation (tpg time) are indicated. All the designs via each of the procedures are 100% testable.

From the standpoint of CPU usage for minimization and test pattern generation the CONSTRAIN procedure used the least time, but required modifying the original design. Unfortunately, circuit modifications which modify interface descriptions, such as adding inputs or outputs, can be expensive (or impossible!) in typical design environments.

The COVER procedure completed all examples with modest to reasonable CPU requirements. The OPTSYN procedure required the greatest amounts of CPU and was prohibitively expensive on one example.

In terms of quality of result, OPTSYN uniformly produced the smallest designs. COVER's results averaged were within 5% over all and were

EX	COVER			CONSTRAIN			OPTSYN		
	lit	syn. time	tpg time	lit	syn. time	tpg time	lit	syn. time	tpg time
ex1	48	5.1s	2.0s	59	4.1s	1.1s	43	11s	2.0s
ex2	60	38s	40s	65	30s	10s	54	1.1m	41.8s
bbara	68	21s	1.6m	84	21s	11s	68	21s	1.8m
bbsse	124	1.3m	2.7m	145	1.3m	17s	124	1.3m	3.2m
s1	154	1.2m	4.6m	175	1.2m	5.5s	141	2.3m	5.1m
dfile	194	1.4m	3.9m	234[1]	1.3m	45s	165	8.2m	5.5m
plan	532	2.6m	2.1m	568	2.6m	54s	532	2.6m	2.3m
scf	794	4.1m	7.3m	852	4.1m	81s	794	4.1m	82m
fs1	451	3.7m	11m	544[1]	2.8m	39s	423	1.3h	26m
fs2	578	5.0m	19m	667[1]	3.9m	51s	-	> 2h[2]	-

[1] Involves the addition of an extra input and output.
[2] The synthesis procedure was terminated after 2 hours.

the precisely the same as OPTSYN/s on four examples. COVER's results on area were uniformly superior to the CONSTRAIN procedure resulting in an average 13% improvement.

Overall, these results indicate that the COVER procedure improves over the previous procedures from the standpoint of quality of result versus CPU time requirements. Most importantly the COVER procedure is able to handle designs that the previous procedures could not (without modification).

These results show that a synthesis user seeking complete testability presently has a spectrum of methods at his disposal, and may choose his approach based on the peculiarities of the example to be synthesized and the relative importance of synthesis CPU time, TPG time, final circuit size and the difficulty of incorporating circuit modifications into the complete circuit design.

7 Conclusions

A variety of techniques have been proposed to address the problem of synthesizing fully testable sequential machines. At one end of the spectrum there are optimal synthesis procedures that ensure full testability by eliminating redundancies via the use of appropriate don't care sets. At the other end of the spectrum there are constrained synthesis procedures that produce fully and easily testable sequential circuits by restricting the implementation of the logic. In this paper we attempted to unify and extend these methods. We first identified classes of redundancies and isolated *equivalent-state redundancies* as those most difficult to eliminate. We then showed that the essential problem behind equivalent-state redundancies is the creation of valid/invalid state pairs. We devoted the remainder of the paper to techniques for developing *differentiating sequences* for valid/invalid state pairs created by a fault, as well as to techniques for retaining these sequences in the presence that fault. We showed how both optimal and constrained synthesis procedures ensure differentiating sequences and also used the notion of *fault-effect disjointness* to demonstrate a spectrum of methods that place relatively more-or-less emphasis on either logic optimization or constrained synthesis. Techniques used in this exploration included fault simulation, Boolean covering, algebraic factorization and state assignment.

We then compared the final results of each of these methods on a number of standard benchmark examples and showed that each approach has its merits depending on the relative importance of synthesis CPU time, TPG time, final circuit size and the difficulty of incorporating circuit modifications into the complete circuit design.

8 Acknowledgements

Thanks to Robert Brayton, Tim Cheng, Tony Ma, Richard Newton, Alex Saldanha and Alberto Sangiovanni-Vincentelli for interesting discussions on sequential circuit optimization and testability. This work was supported in part by the Defense Advanced Research Projects Agency under contract N00014-87-K-0825.

References

[1] K. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Multi-level Logic Minimization Using Implicit Don't Cares. In *IEEE Transactions on Computer-Aided Design*, pages 723-740, June 1988.

[2] D. Brand. Redundancy and don't cares in logic synthesis. In *IEEE Transactions on Computers*, volume C-32, pages 947-952, October 1983.

[3] R. Brayton. Factoring logic functions. In *IBMJRD*, pages 187-198, March 1987.

[4] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A Multiple-Level Logic Optimization System. In *IEEE Transactions on Computer-Aided Design*, pages 1062-1081, November 1987.

[5] K-T. Cheng and V. D. Agrawal. An Economical Scan Design for Sequential Logic Test Generation. In *Proc. of 19th Fault Tolerant Computing Symposium*, June 1989, to appear.

[6] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Optimal Logic Synthesis and Testability: Two Faces of the Same Coin. In *Proc. of International Test Conference*, pages 3-13, September 1988.

[7] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. A Synthesis and Optimization Procedure for Fully and Easily Testable Sequential Machines. In *IEEE Transactions on Computer-Aided Design*, October 1989, to appear.

[8] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Irredundant Sequential Machines Via Optimal Logic Synthesis. In *Proc. of 23rd Hawaii Conference on System Sciences and to appear in IEEE Transactions on Computer-Aided Design*, January 1990.

[9] G. D. Hachtel, R. M. Jacoby, K. Keutzer, and C. R. Morrison. On the Relationship Between Area Optimization and Multifault Testability of Multilevel Logic. In *Proceedings of the International Workshop on Logic Synthesis*, June 1989.

[10] E. J. McCluskey. Minimization of Boolean Functions. In *Bell Lab. Technical Journal*, volume 35, pages 1417-1444, Bell Lab., November 1956.

[11] A. Miczo. *Digital Logic Testing and Simulation*. Harper and Row, New York, 1986.